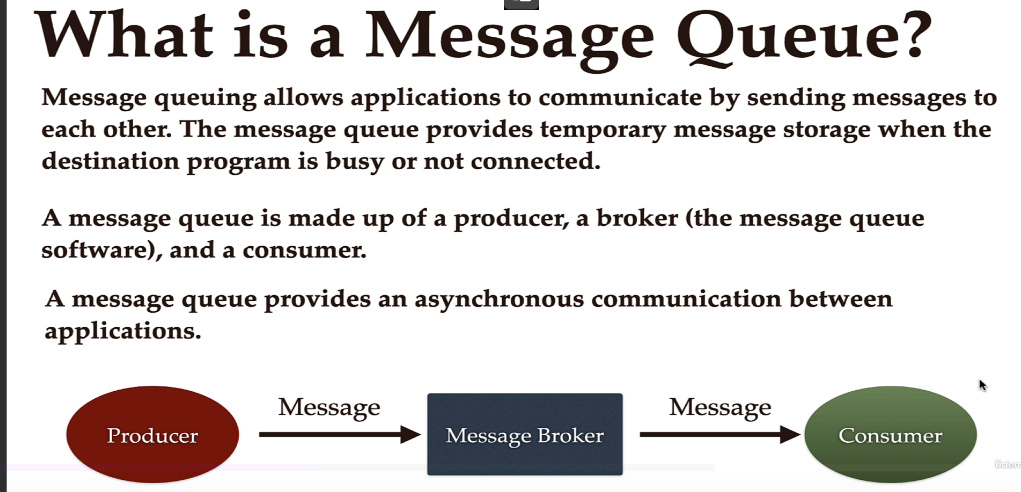
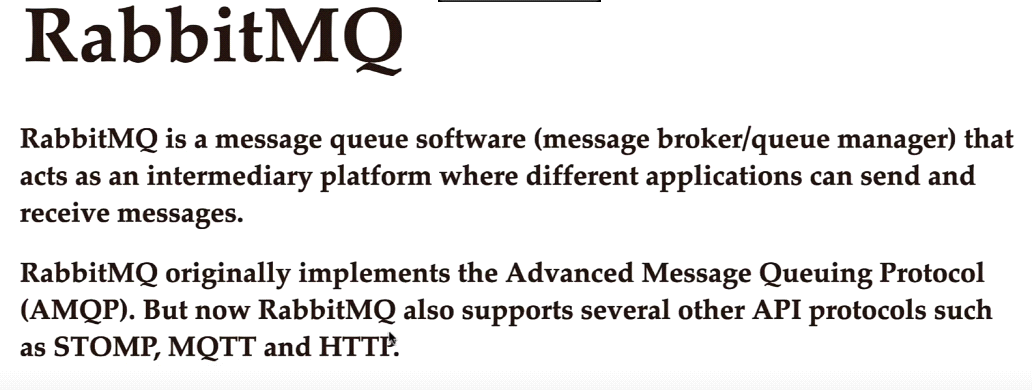
* RabbitMQ poate fi folosit in proeiect cu ajutorul lui Spring AMQP
* Spring AMQP - Spring Advanced Message Queuing Protocol

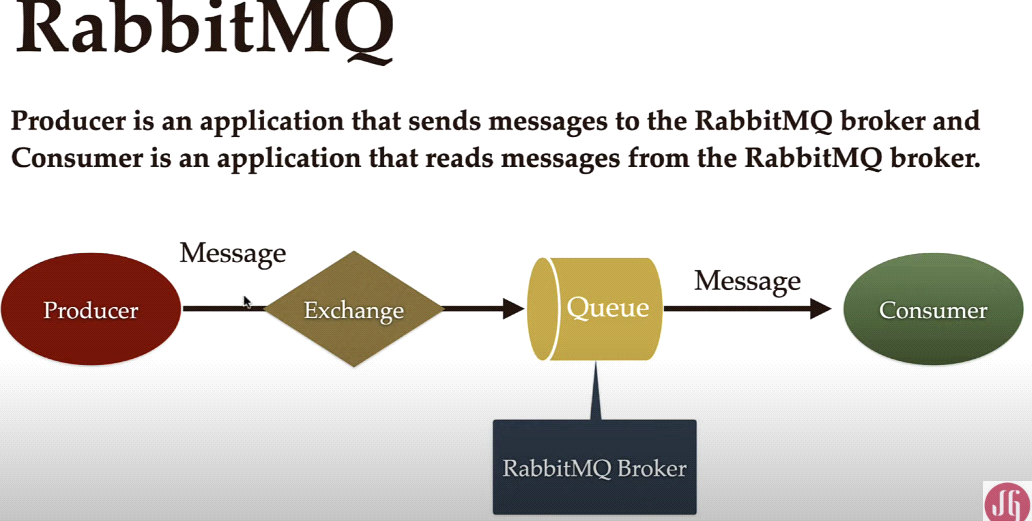
**Message Queue/Message Broker**

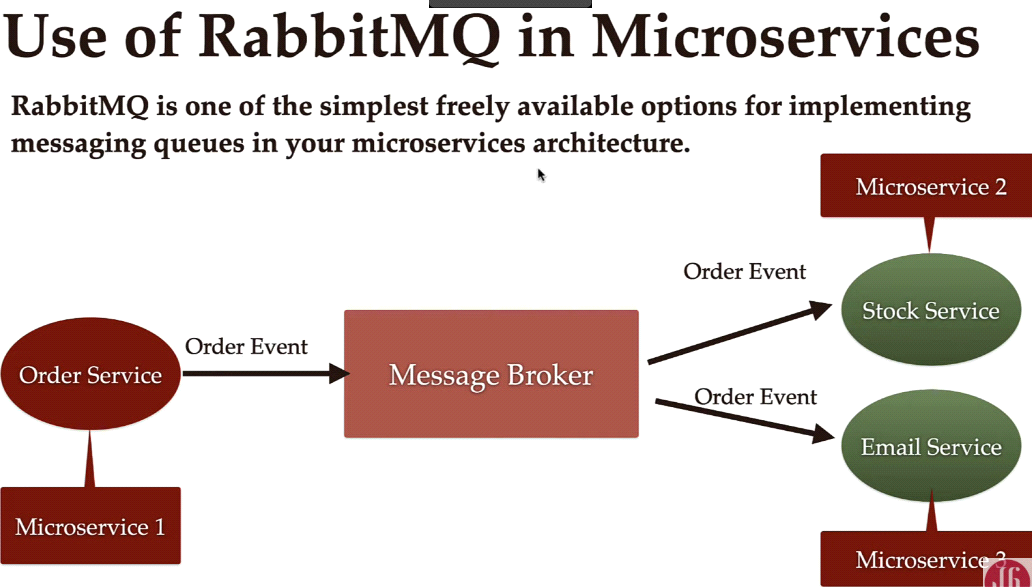
* Message Queue != Message Broker!!!
* Message queue este doar o structura de a pastra mesajele, o componenta din Message Broker
* Message broker permite aplicatiilor sa comunice intre ele prin a trimite mesaje una la alta. Message queue ofera temporary message storage cand destination program este ocupat sau neconectat
* Un message broker ofera comunicare asyncronizata intre aplicatii

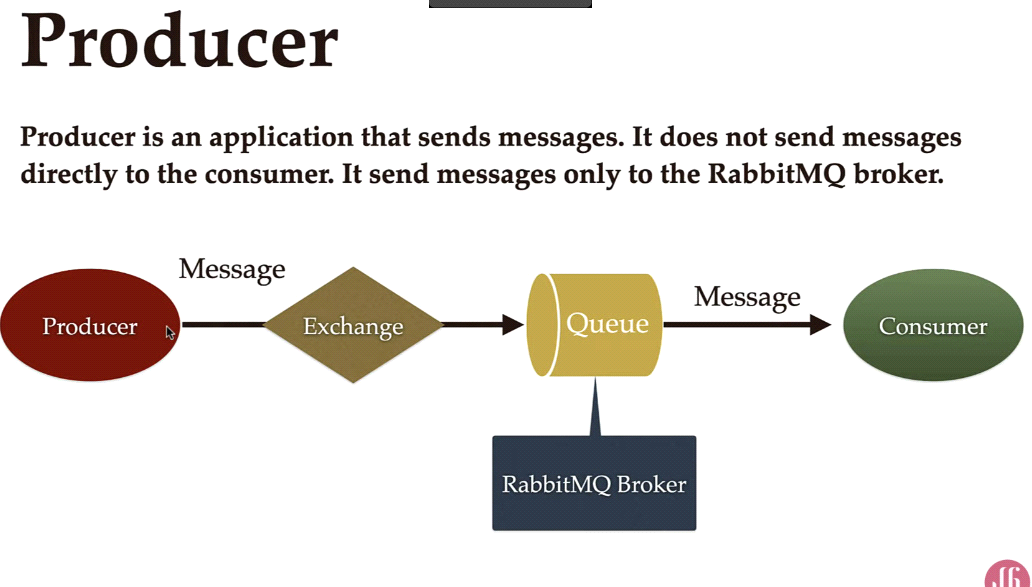


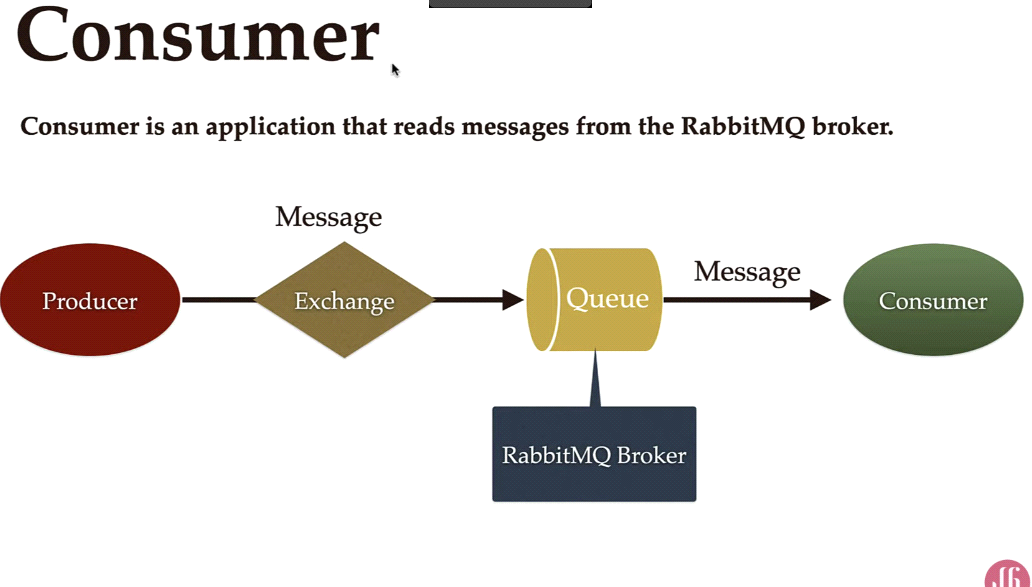
**RabbitMQ**



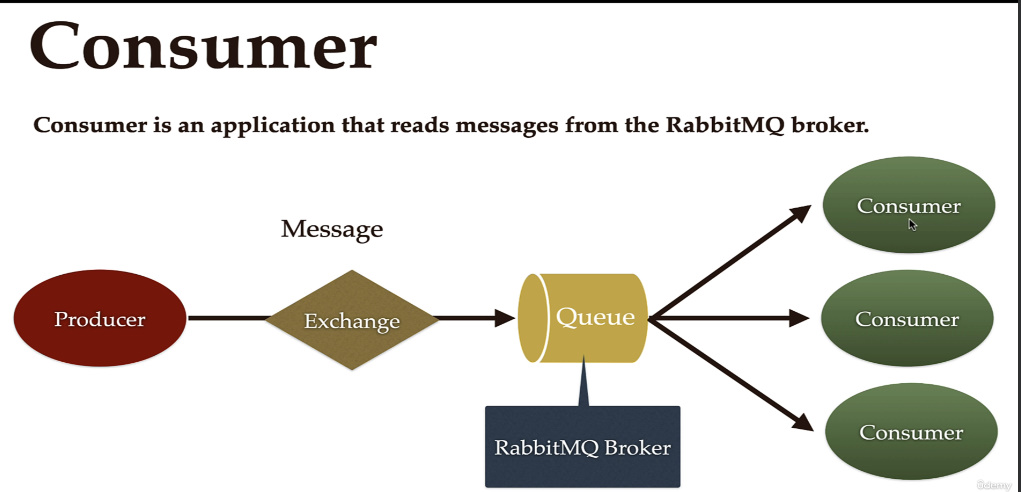




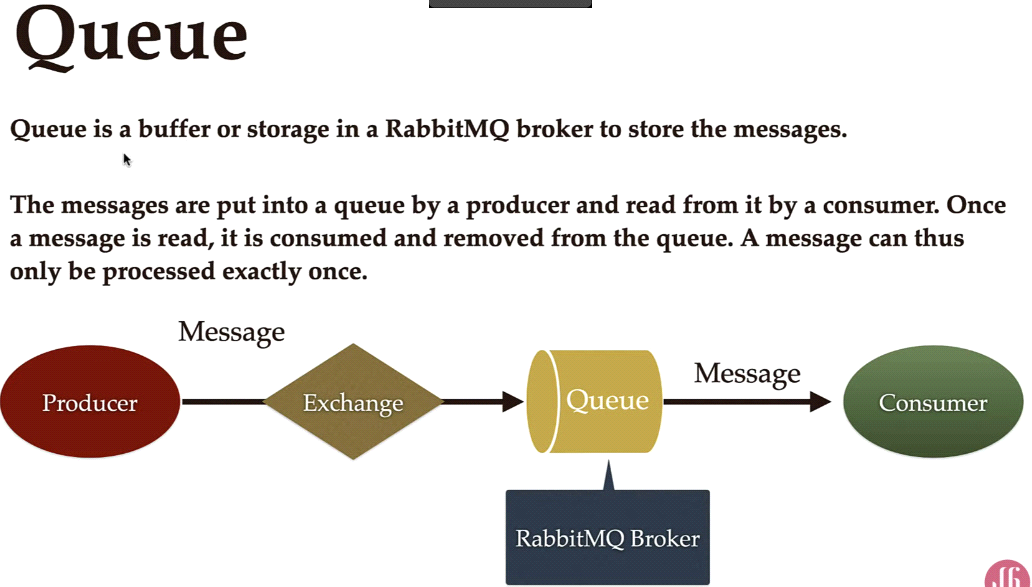




Pot fi si mai multi consuemrs care preiau mesaje de la un message queue:

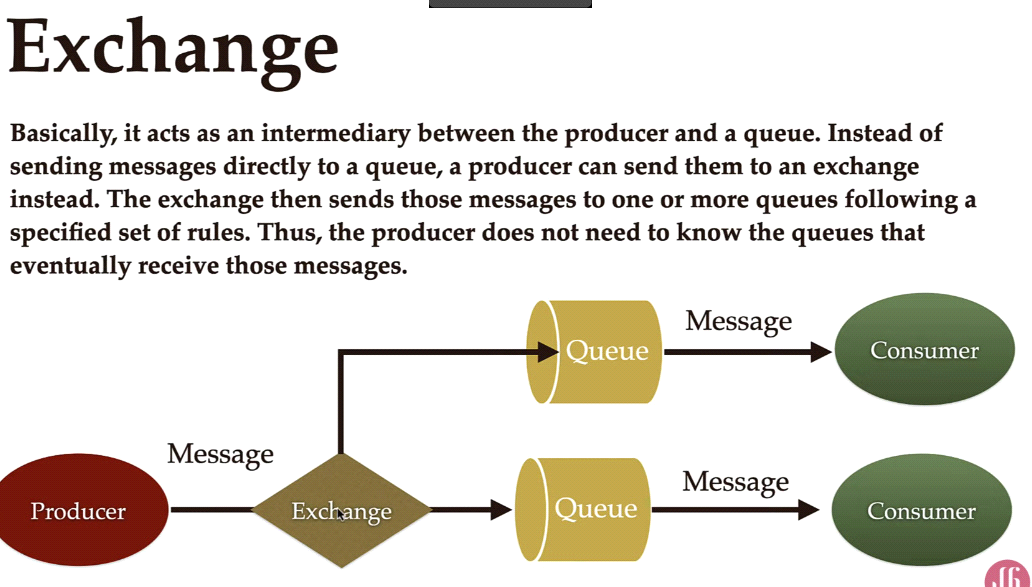


Toti consuemers vor primi mesajul. In kafka, toate groups consumers primesc mesajul, in RabbitMQ nu exista group consumer, ci direct consumers.



Queue este o componenta din RabbitMQ Broker

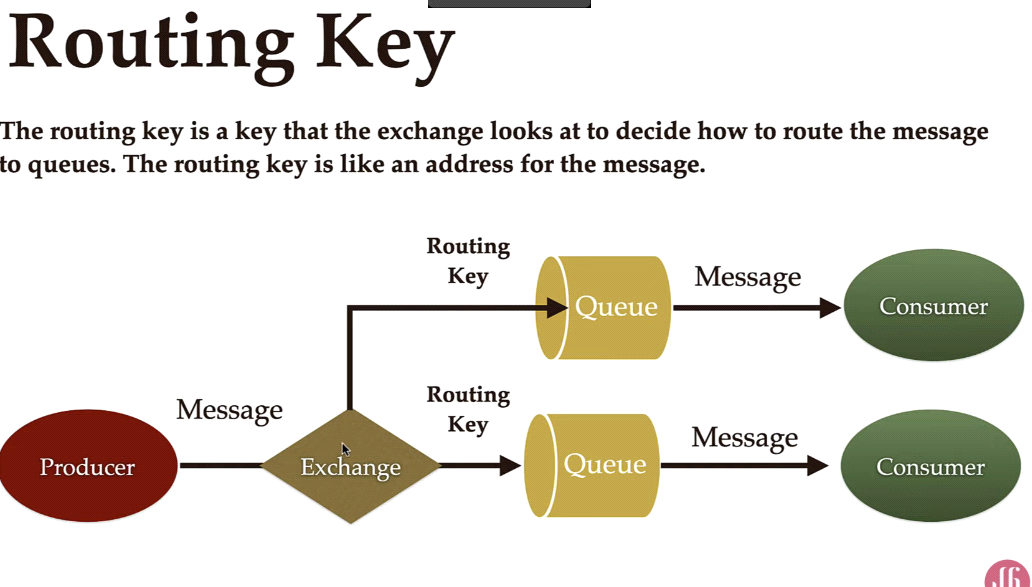
Un RabbitMQ Broker poate avea mai multe queues, asa cum Kafka are mai multe topics



Exchange e cel care asigura transportul mesajului de la Producer la Queue.

In Kafka, se trimite key + value message. Cand cream consumer, noi decidem in ce topic sa fie pus, si optional putem decide si in ce partition sa fie pus. Anume key in kafka arata in ce partition sa fie pus, si daca nu noi o specificam, consumer singur alege. In RabbitMQ tot exista asa sistem si el e numit Exchange. Daca nu oferim numele la queue, consumer in rabbitmq creaza un routing key ce va fi queue, sau daca il oferim noi, acela si va fi routing key

In kafka trebuie neaparat sa scriem topic in care sa se duca message



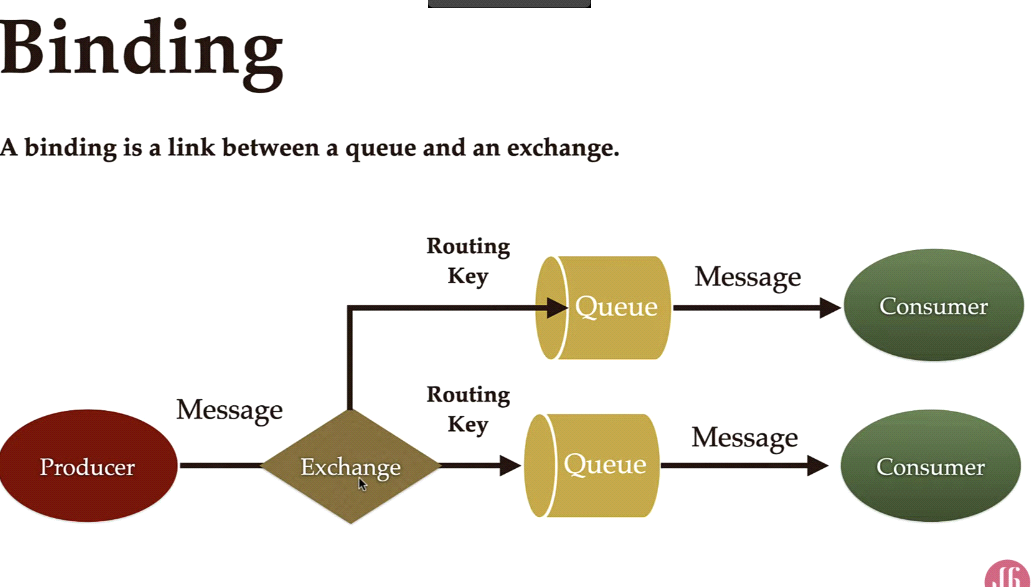
- Routing key este creat de producer si e trimis impreuna cu message, asa cum face kafka cu key : value

- Routing key e pur si simplu un string ce spune in ce queue trebuie pus message si gata, gen routing key poate fi “queue1” sau “1”

- Cand Producer trimite un mesaj la Exchange, Exchange inca nu stie la care queue el trebuie trimis

- Ecxhange foloseste routing key pentru a scrie mesajul in queue respectiv

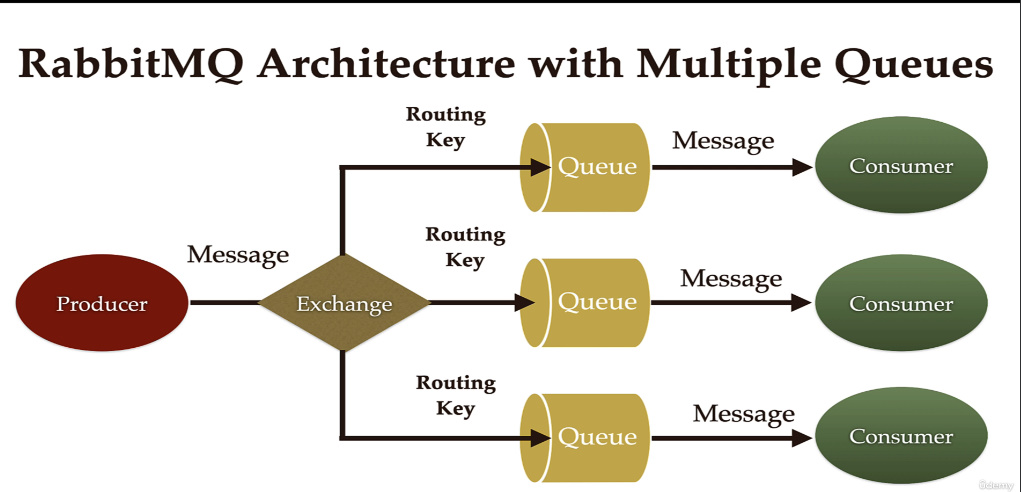
- routing key e trimis si creat tot de producer catre exchange, caci exchange inca nu poate face conexiunea fara routing key



Iar acest binding e facut de Routing Queue

De ex, producer vrea sa trimita un mesaj la Queue2. El va trebui sa trimita mesajul la exchange si routing key pentru queue si exchange va folosi acest routing key ca sa identifice queue2, sa faca conexiunea si sa trimita mesajul, deci va face conexiunea cu queue cerut, adica queue2

**Architecure with multiple queues**



**Configuratie**

* Instalam in docker imaginea oficiala, anume 3.12.0-management, management e pentru ui interface
* port din web: 15672
* port pentru conexiuni: 5672
* default login and password: guest guest
* Putem folosi si:

**docker run -it -p 5672:5672 -p 15672:15672 -e RABBITMQ\_DEFAULT\_USER=guest -e RABBITMQ\_DEFAULT\_PASS=guest rabbitmq:3-management**

**si ne conectam cu localhost:15672**

**Web interface**

* Connection - conexiunile facute de aplicatiile noastre cu rabbitmq
* Channels - aici apar mesajele intre producer si consumer
* Exchanges - e important sa cream un exchange

Un exchange trebuie legat de un queue

1. Cream un exchange

2. cream un queue Classic

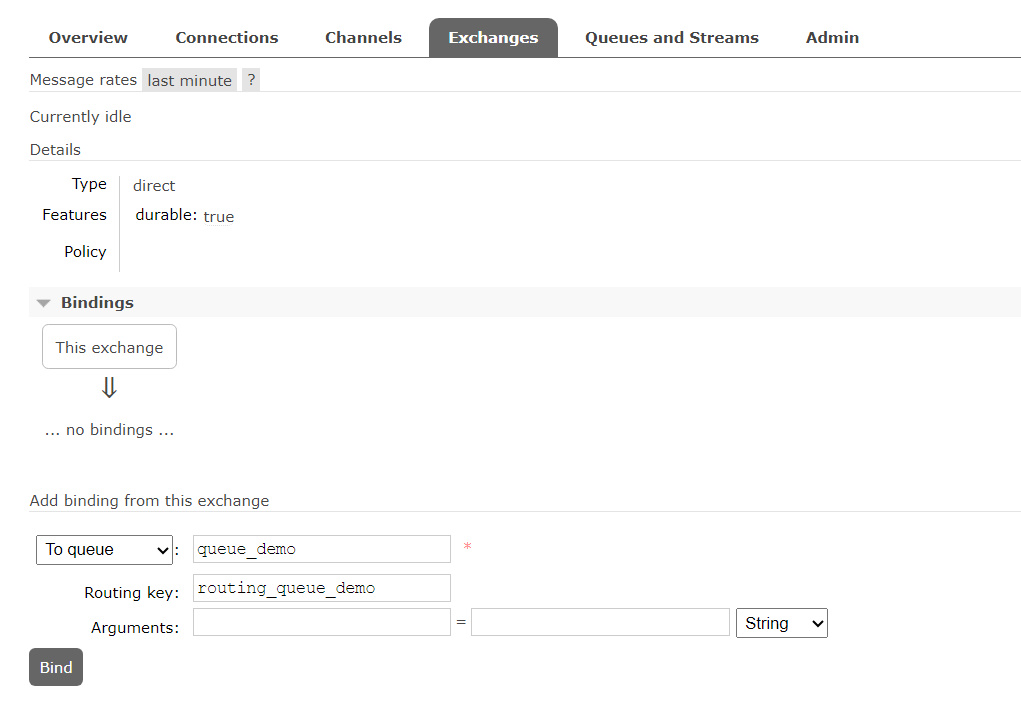
3. Acum, ne ducem la exchange creat si il legam de queue creat prin a crea un binding

La routing punem ce nume vrem

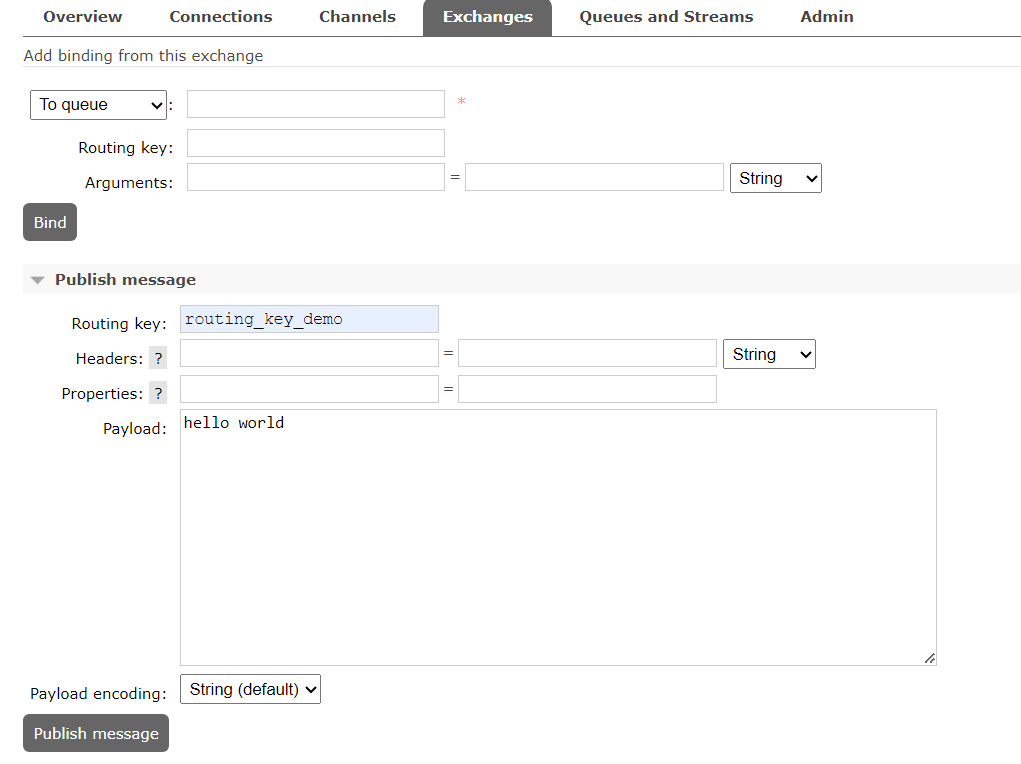
Exchangee trimite mesajele la Queue, si asta o putem face manual si din exchange din web

routing key e un fel de index ce il vom folosi pentru a trimite la queue\_demo

queue nici nu stie de acest routing key, doar exchange stie, si prin folosirea lui, el poate sti la care queue anume sa trimita un mesaj.

4. La queue dam Get Messages si vedem mesajele venite. Putem trimite direct un message din exchange: 

si il vedem aici



**Spring Boot**

* Spring AMQP este doar o colectie de interfete ce trebuie implementate. Spring RabbitMQ o implementeaza. Deci Spring AMQP este un API pentru sytemele ce folosesc AMQP protocol
* Ne trebuie dependenta:

<dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-amqp</artifactId>  
</dependency>

Ea ofera si implementare, care e rabbitmq by default

* Acum, configuram rabbitmq:

spring.rabbitmq.host = localhost  
spring.rabbitmq.port = 5672  
spring.rabbitmq.username = guest  
spring.rabbitmq.password = guest

* Acum, ne va trebui sa cream beanuri pentru rabbitmq queue:

Dependentele le luam din org.springframework.amqp.core.

* Acum cream un bean si pentru exchange:

**@Bean  
public Queue queue(){  
 return new Queue("queue\_demo");  
}  
Importurile sa fie de la**

import org.springframework.amqp.core.

@Bean  
public TopicExchange topicExchange(){  
 return new TopicExchange("exchange\_demo");  
}

* Trebuie sa facem binding la acest queue si exchange folosind routing key setat:

@Bean  
public Binding binding(){  
 return BindingBuilder.*bind*(queue())  
 .to(topicExchange())  
 .with("routing\_key\_demo");  
}

Avem grija sa fie numele identic cu cele setate in Web, caci aici facem un fel de mapare. Daca un queue sau binding sau exchange cu asa nume nu exista, se va crea, dar abea cand se va trimite un mesaj, nu deodata. Daca deja exista, crearea acestor beanuri nu mai e necesara.

* In mod normal ne-ar mai trebui beanuri de tip ConnectionFactory, RabbitTemplate, RabbitAdmin, dar Spring Boot deja le configureaza automat

**Trimiterea si primirea mesajelor**

* RabbitTemplate este beanul care ne ofera posibilitatea de a primit si trimite mesaje rapid. El e creat automat de Spring Boot
* Acum putem crea un bean care sa trimita mesajele**(producer),** gen un @Service

@RequiredArgsConstructor  
@Service  
public class RabbitMqService {  
 private RabbitTemplate rabbitTemplate;  
  
 public void sendMessage(String message){  
 rabbitTemplate.convertAndSend("exchange\_demo","routing\_key\_demo","message");  
  
 System.*out*.println("Message send to queue\_demo");  
 }  
  
}

**convertAndSend("exchange name","routing key name","message")** - mesajul e convertit automat in Message si e trimis la exchange cu routing key.

Acum facem res API:

@RestController  
@RequiredArgsConstructor  
public class Rest {  
 private final RabbitMqService rabbitMqService;  
  
 @GetMapping("/api/{message}")  
 public String get(@PathVariable("message")String msg){  
 rabbitMqService.sendMessage(msg);  
  
 return "Message send to rabbitmq";  
 }  
  
}

**Atentie! Beanurile la Queue, exchange si bind se fac doar cand vrem sa cream un queue, exchenger si routing key dintre ele nou. Daca ele deja exista, nu mai trebuie beanuri.**

**Create Consumer**

@Service  
public class RabbitMqConsumer {  
  
 @RabbitListener(queues = {"queue\_demo"})  
 public void listener(String message){  
 System.*out*.println("Message recceived: "+message);  
 }  
}

Punem doar @RabbitListener cu numele la queues si gata

**Multiples consumer**

* Daca de ex cream mai multe consumers conectate la aceeasi queue, asa:
* @Service  
  public class RabbitMQConsumer {  
   private final Logger logger = Logger.*getLogger*(RabbitMQConsumer.class.getName());  
   @RabbitListener(queues = {"${spring.rabbitmq.queue1}"})  
   public void consumeMessage1(String message){  
   logger.info(String.*format*("Message received by consumer1 => "+message));  
   }  
   @RabbitListener(queues = {"${spring.rabbitmq.queue1}"})  
   public void consumeMessage2(String message){  
   logger.info(String.*format*("Message received by consumer2 => "+message));  
   }  
    
  }

Mesajul trimis se va duce doar la un singur consumer!!! Chiar daca sunt 2 consumers la aceeasi queue, mesajul va fi citit doar de unul!

* In kafka, fiecare group consumer citeste mesajul, in RabbitMQ doar un consumer poate citi mesajul
* Insa, RabbitMQ va face un balance. Intai consumer1 va citit un mesaj, apoi urmatorul mesaj va fi citit de consumer2 si tot asa.

**JSON Message Configuration**

Un exchange poate avea diferite routing keys, deci un exchange poate crea legaturi cu mai multe queues

* Cream o noua routing key pentru json
* Cream un RabbitTemplate si setam un MessageConvertor in JSON:

@Bean  
public RabbitTemplate rabbitTemplate(ConnectionFactory connectionFactory){  
 RabbitTemplate rabbitTemplate = new RabbitTemplate(connectionFactory);  
 rabbitTemplate.setMessageConverter(new Jackson2JsonMessageConverter());  
   
 return rabbitTemplate;  
}

Acum, modificam metoda de a trimite mesajul, sa trimita un user, desi el tot v-a fi trimis ca String, doar ca scris ca JSON

@RequiredArgsConstructor  
@Service  
public class RabbitMqService {  
 private final RabbitTemplate rabbitTemplate;  
  
 public void sendMessage(User user){  
 rabbitTemplate.convertAndSend("exchange\_demo","routing\_key\_demo\_json",user);  
  
 System.*out*.println("User sent to queue");  
 }  
  
}

Si modificam putin Controller:

@GetMapping("/api/{message}")  
public String get(@PathVariable("message")String msg){  
 User user = new User(1,"Test","User");  
 rabbitMqService.sendMessage(user);  
  
 return "Message send to rabbitmq";  
}

Si, daca scoatem acest listener, vedem ca s-a inregistrat useru in Rabbit

* Acum, pentru a primi json si a-l converti in object:

@RabbitListener(queues = {"queue\_demo"})  
public void listener(User user){  
 System.*out*.println("User recceived: "+user);  
}

Atentie! RabbitMQ va converti singur din JSON in User, dar el are nevoie de un Bean de tip Jackson2JsonMessageConverter

Noi l-am creat direct in RabbitTemplate bean, dar cand va deserializa, el va cauta asa bean si daca nu il gaseste, avem exceptie,de aceea cream beanul:

@Bean  
public MessageConverter messageConverter(){  
 return new Jackson2JsonMessageConverter();  
}

**@RabbitHandler**

* Este exact ca @KafkaHandler. Creaza un un consumer ce poate consuma obiecte de diferite tipuri
* @Service  
  @RabbitListener(queues = {"${spring.rabbitmq.queue2}"})  
  public class RabbitMQJSONConsumer {  
   @RabbitHandler  
   public void consumeUser(UserDto userDto){  
   System.*out*.println("Consumed JSON user => "+userDto);  
   }  
   @RabbitHandler  
   public void consumeProduce(ProduceDto produceDto){  
   System.*out*.println("Consumed JSON produce => "+produceDto);  
   }  
  }